(19) Europäisches Patentamt
European Patent Office
Office européen des brevets

(11) EP 3 264 307 A1

(12) EUROPEAN PATENT APPLICATION

(43) Date of publication:
03.01.2018 Bulletin 2018/01

(51) Int Cl.:
G06F 21/14 *(2013.01)*

(21) Application number: **16305797.9**

(22) Date of filing: **29.06.2016**

(84) Designated Contracting States:
AL AT BE BG CH CY CZ DE DK EE ES FI FR GB GR HR HU IE IS IT LI LT LU LV MC MK MT NL NO PL PT RO RS SE SI SK SM TR
Designated Extension States:
BA ME
Designated Validation States:
MA MD

(71) Applicant: **Nagravision SA**
1033 Cheseaux-sur-Lausanne (CH)

(72) Inventors:
- **DORÉ, Laurent**
  35235 Thorigné-Fouillard (FR)
- **PIRET, Eric**
  35830 Betton (FR)
- **WYSEUR, Brecht**
  1305 Panthalaz (BE)
- **BELAIDI, Yasser**
  35770 Verne-sur-Seche (FR)

(74) Representative: **Thorniley, Peter**
Kilburn & Strode LLP
Lacon London
84 Theobalds Road
London WC1X 8NL (GB)

(54) ON DEMAND CODE DECRYPTION

(57) A system and a method for protecting code are provided. Extraction of code to be protected takes place during an object-to-object transformation and that code is replaced with fake binary code. The extracted code to be protected may then be encrypted or otherwise obscured and stored in a separate region of an object file. A prior source-to-source file transformation can be provided to isolate and mark the code to be protected, and to inject additional source code to handle later decryption.

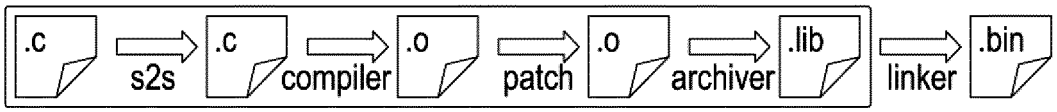

Fig. 2B

EP 3 264 307 A1

Description

FIELD

[0001] The present disclosure relates to a system and method for protecting code, for example by adopting a build process which facilitates on demand code decryption.

BACKGROUND

[0002] Software can be subject to malicious attack by external parties, such as reverse engineering attacks. In view of this, various techniques have been developed to protect software from such attacks.

[0003] An example of such a technique is known as "on-demand code decryption". According to this technique, some elements, or "chunks", of the code are delivered in an encrypted form. These are decrypted just prior to execution and then purged afterwards. This can in particular mitigate static analysis techniques which examine the code without executing it. Static analysis techniques include multiple variations and typically involve disassembling machine code.

[0004] Typically, on-demand encryption processes can be broadly summarised as comprising four steps. Firstly, the relevant binary code to be protected is extracted. Secondly, fake code is substituted in position of the protected binary code. Thirdly, the extracted binary code is then encrypted and added to a data section of the binary. The final binary is then finalized in such a way that the process of on-demand decryption is provided with the correct information to use. Conventionally, these steps are each carried out after the process of linking has been completed, and are thus carried out by the integrator.

BRIEF DESCRIPTION OF THE DRAWINGS

[0005]

FIG.1 shows a hardware infrastructure for implementing a preferred embodiment;

Figure 2A illustrates a compiling process according to a known technique;

Figure 2B illustrates a compiling process according to the preferred embodiment;

Figure 3 illustrates a source-to-source transformation within the compiling process of the preferred embodiment;

Figure 4 illustrates an object-to-object transformation within the compiling process of the preferred embodiment;

Figure 5 shows a flow chart of an on-demand decryption process.

Figure 6 shows areas of the code from processing which are to be excluded as relocation directions;

Figure 7 shows generation of a mask to be applied to protect excluded areas during on-demand code decryption; and

Figure 8 illustrates a process of on-demand code decryption that does not influence relocation addresses.

DETAILED DESCRIPTION OF THE DRAWINGS

[0006] In overview, a system and a method for protecting code are provided. Extraction of code to be protected takes place during an object-to-object transformation and that code is replaced with fake binary code. The extracted code to be protected may then be encrypted or otherwise obscured and stored in a separate region of an object file. A prior source-to-source file transformation can be provided to isolate and mark the code to be protected, and to inject additional source code to handle later decryption.

[0007] In some aspects of the disclosure there is provided a method, for example a method for protecting code such as a computer-implemented method for protecting code. The method comprises carrying out an object-to-object file transformation. The object-to-object transformation comprises identifying code to be protected within an input object file for encryption and extracting the identified code to be protected. The object-to-object transformation further comprises replacing the identified code to be protected within the input object file with a fake code to generate a first output object file and injecting the code to be protected into a second output object file. The code to be protected may be encrypted or obscured using another technique prior to injection into the second output object file.

[0008] By carrying out an object-to-object transformation of this kind, a method may be provided which may avoid a requirement to carry out decryption at a later stage of the compilation process. For example, in contrast with approaches which rely on encryption at the linking stage carried out by the integrator when generating the final binary, the method of the present disclosure can allow code to be obscured without relying on full coordination of the integrator with the processes for generating source and object code. This is a more reliable to solution in many scenarios in which the integrator is independently operated to earlier coding/compiling processes. Improved reliability and security may be achieved by maintaining control of the obscuration process close to the source of the code. The fake code may be selected to resemble real code, thereby making the encryption process more difficult to detect. The identified code to be protected may be injected into a data region of the second output object file, which may further help to obscure its true nature.

[0009] In some embodiments, the first output object file and the second output object file may be consolidated into a final output object file. This may allow a one-to-one relationship between any input object files and output object files such that there is a minimal need to adjust

later processing to take account of modifications during the object-to-object transformation.

[0010] In some embodiments, the method may further comprise, prior to the object-to-object transformation, carrying out a source-to-source transformation. The source-to-source transformation may comprise marking the code to be protected within an input source file. The source-to-source transformation may further comprise providing additional code to the input source file to provide instructions for a later decryption operation. Moreover, the source-to-source transformation may further comprise isolating the code to be protected. In this manner, the source-to-source transformation may allow necessary information for the object-to-object transformation to be integrated without deliberate operator action. Alternatively or additionally, the original source code may be provided with appropriate information.

[0011] In some embodiments, the object-to-object transformation may further comprise: identifying relocation directions within the input object file; excluding the relocation directions from replacement by the fake code. The relocation directions may be addresses to which relocations are addressed. By excluding these from the code to be protected, the process of relocation during the linking stage may be unaffected by the method. In effect, this may allow the code to be protected to call external functions at the linker/integrator despite not being visible to the linker/integrator since it is encrypted at that stage. In some embodiments, instead of excluding the relocation directions from replacement, the object-to-object transformation may comprise altering the target of one or more relocation directions. This may ensure the consistency of the decrypted code.

[0012] In some aspects of the disclosure, a computer program product is provided comprising computer executable instructions for carrying out the method described above. In further aspects, a system is provided for carrying out the method described above.

[0013] Some specific embodiments are now described by way of illustration with reference to the accompanying drawings in which like reference numerals refer to like features.

[0014] Figure 1 illustrates a block diagram of one implementation of a computing device 100 within which a set of instructions, for causing the computing device to perform any one or more of the methodologies discussed herein, may be executed. In alternative implementations, the computing device may be connected (e.g., networked) to other machines in a Local Area Network (LAN), an intranet, an extranet, or the Internet. The computing device may operate in the capacity of a server or a client machine in a client-server network environment, or as a peer machine in a peer-to-peer (or distributed) network environment. The computing device may be a personal computer (PC), a tablet computer, a set-top box (STB), a Personal Digital Assistant (PDA), a cellular telephone, a web appliance, a server, a network router, switch or bridge, or any machine capable of executing a set of instructions (sequential or otherwise) that specify actions to be taken by that machine. Further, while only a single computing device is illustrated, the term "computing device" shall also be taken to include any collection of machines (e.g., computers) that individually or jointly execute a set (or multiple sets) of instructions to perform any one or more of the methodologies discussed herein.

[0015] The example computing device 100 includes a processing device 102, a main memory 104 (e.g., read-only memory (ROM), flash memory, dynamic random access memory (DRAM) such as synchronous DRAM (SDRAM) or Rambus DRAM (RDRAM), etc.), a static memory 106 (e.g., flash memory, static random access memory (SRAM), etc.), and a secondary memory (e.g., a data storage device 118), which communicate with each other via a bus 130.

[0016] Processing device 102 represents one or more general-purpose processors such as a microprocessor, central processing unit, or the like. More particularly, the processing device 102 may be a complex instruction set computing (CISC) microprocessor, reduced instruction set computing (RISC) microprocessor, very long instruction word (VLIW) microprocessor, processor implementing other instruction sets, or processors implementing a combination of instruction sets. Processing device 102 may also be one or more special-purpose processing devices such as an application specific integrated circuit (ASIC), a field programmable gate array (FPGA), a digital signal processor (DSP), network processor, or the like. Processing device 102 is configured to execute the processing logic (instructions 122) for performing the operations and steps discussed herein.

[0017] The computing device 100 may further include a network interface device 108. The computing device 100 also may include a video display unit 110 (e.g., a liquid crystal display (LCD) or a cathode ray tube (CRT)), an alphanumeric input device 112 (e.g., a keyboard or touchscreen), a cursor control device 114 (e.g., a mouse or touchscreen), and an audio device 116 (e.g., a speaker).

[0018] The data storage device 118 may include one or more machine-readable storage media (or more specifically one or more non-transitory computer-readable storage media) 128 on which is stored one or more sets of instructions 122 embodying any one or more of the methodologies or functions described herein. The instructions 122 may also reside, completely or at least partially, within the main memory 104 and/or within the processing device 102 during execution thereof by the computer system 100, the main memory 104 and the processing device 102 also constituting computer-readable storage media.

[0019] For comparative purposes, Figure 2A illustrates the general steps of a known software build process. Here, source files ".c" are compiled by a compiler to create object files ".o". The object files are then archived by an archiver to form libraries ".lib" which are in turn linked

by a linker (sometimes referred to as an integrator) to form a final binary file ".bin". Each of the complier, archiver and integrator may be implemented on a computing device 100 such as that described in Figure 1. The archiver, compiler and integrator may each be implemented on an independent computing device 100, or any combination of the archiver, compiler and integrator may share a computing device upon which they are implemented. For example, the compiler and archiver may be integrated on a first computing device 100 and the integrator may be implemented on a second computing device 100. Where multiple computing devices 100 are provided, these may communicate over any appropriate communications network.

[0020] In many conventional scenarios the compiler and archiver may be under control of a first entity, while a second entity may aggregate libraries from multiple sources through implementation of a linker/integrator. Box 210 in Figure 2A illustrates the entities under control of the first entity. Accordingly, when the binary .bin file is produced, multiple entities have had access to the code, increasing potential security risks as well as stability risks where coordination between entities is imperfect. The present disclosure provides a build process which mitigates such risks. In particular, this build process enables on-demand code decryption. Such a build process is illustrated in Figure 2B.

[0021] In comparison to Figure 2A, Figure 2B illustrates two additional build steps. Firstly, an "s2s" source-to-source transformation transforms an input source file .c into an output source file .c. This is then compiled by a compiler to form an object file in line with the process of Figure 2A. This object file is then an input object file for an object-to-object transformation labelled as "patch" in Figure 2B, which generates one or more output object files.

[0022] The source-to-source transformation can be understood with reference to Figure 3, which illustrates an example of such a process. In particular, the source-to-source transformation isolates and marks the code to be protected with markers. The operation "fibWrapped" identifies this code. Additionally, during this transformation additional code is incorporated to assist in handling the decryption operation. In some examples, an alternative process to encryption may be used to obscure the code to be protected, in which case the decryption operation will be replaced by a suitable alternative.

[0023] Figure 4 illustrates an example of the object-to-object transformation. Here input object file fib.s2s.o contains markers "fibWrapped" and "fibWrappedEnd" allowing the object-to-object transformation to identify the code to be protected. This code is extracted and replaced with fake code within the object file fib.s2s.o. The fake code can be selected to resemble real code, and may be, for example, junk code, real code or seemingly meaningful code. In other examples, the fake code may be random code. The modified object file fib.s2s.o may be considered a first output object file.

[0024] In addition, the object-to-object transformation may generate an intermediate source file fib.shellcode.c. This intermediate source file is used to encrypt the code to be protected using an encryption operation matching the decryption operation injected during the source-to-source transformation and a give secret key. The secret key may be predefined or may be defined such that it can be derived during the object-to-object transformation or at another time. The encryption operation may be replaced with an alternative form of obfuscation, which may be weaker, in order to minimize overheads and potential performance penalties.

[0025] The intermediate source file is compiled during the object-to-object transformation to generate a second output object file, referred to as "fib.shellcode.o" in Figure 4. The second object file carries the encrypted or otherwise obscured code to be protected in a data section.

[0026] The first and second object files may subsequently be consolidated to form a single, consolidated output object file, although this is not required in all embodiments. In this manner, a one-to-one relationship may be achieved between object files used as an input to the object-to-object transformation and those that are output from this process. The approach to object file consolidation will vary in dependence on the toolchain. In some examples, a COFF-format parser may be developed and the process may involve reading both object files, consolidating them according to the Microsoft COFF specification and writing the consolidated file back to disk. In other examples, there may be toolchain-provided tools to carry out this consolidation.

The object file(s) generated by the process of Figures 2B, 3 and 4 can then be passed to an integrator/linker for linking. The integrator does not need to take further steps to ensure that on-demand decryption is possible and does not need to carry out any post-link process. Furthermore, since the code delivered to the integrator is already encrypted, static analysis of the library at this stage is inhibited, increasing the security of the code.

[0027] Figure 5 illustrates a process of on-demand decryption subsequently carried out when the software is run. Firstly, ultimate .bin binary wrapper is obtained at step s51 and the relevant function code (i.e. the code that has been protected) can be retrieved. This is decrypted at step s52 and then patched at step s53 into its source location, replacing the fake code that had been located there.. The program may then be run, at step s54. Subsequently, the function code is unpatched at step s55, once again obscuring this code from static analysis.

[0028] During the patching step s53, certain areas may be preserved, particularly areas modified by the linker after encryption was completed. An example process will be described in more detail below with reference to Figures 6 to 8. At link step, the linker modifies the code, updating offsets in CALL instructions to the relevant target functions. As this cannot be pre-computed in the encrypted code, in this approach described below with ref-

erence to Figure 6 to 8, these are anticipated, such areas are identified prior to encryption, and then preserved so that the result after patching is a proper, correct code. An alternative approach might involve a process of obfuscation OBF, and its symmetric UNOBF, that would work with the linker so that LINK(area) = UNOBF(LINK(OBF(area)); this alternative may avoid the requirement to preserve areas.

[0029] As mentioned above, further details of some preferred embodiments are illustrated in Figures 6 to 8. Here it is recognized that relocation processes may require unaltered code. Such relocation processes can occur during the linking process and when the program is loaded. In order to avoid interference with this process, during the object-to-object transformation areas which are used for relocation directions can be excluded from replacement by the fake code. In particular, the areas used for relocation directions may be areas targeted by relocation commands.

[0030] Figure 6 illustrates an example. The original, "plain" code in the input object file includes two highlighted regions which are the target of relocation operations. The fake code is modified so that these regions are not replaced and values remain constant in these regions.

[0031] A mask may then be generated to ensure any data provided to the regions during relocation is not overwritten during the on-demand decryption process at runtime. The mask may be generated by comparison of the (decrypted) code to be protected and the equivalent area within the output object file. This is illustrated in Figure 7; an XOR operation identifies where the two sets of code are identical, thus indicating where no substitution has been made.

[0032] Relocations occur during linking and loading processes, as illustrated in Figure 8. The mask is then utilized to ensure that during the on-demand decryption process these relocations remain effective by inhibiting the patching of the code that has been protected into the regions reserved for such relocations.

[0033] The various methods described above may be implemented by a computer program. The computer program may include computer code arranged to instruct a computer to perform the functions of one or more of the various methods described above. The computer program and/or the code for performing such methods may be provided to an apparatus, such as a computer, on one or more computer readable media or, more generally, a computer program product. The computer readable media may be transitory or non-transitory. The one or more computer readable media could be, for example, an electronic, magnetic, optical, electromagnetic, infrared, or semiconductor system, or a propagation medium for data transmission, for example for downloading the code over the Internet. Alternatively, the one or more computer readable media could take the form of one or more physical computer readable media such as semiconductor or solid state memory, magnetic tape, a removable computer diskette, a random access memory (RAM), a read-only memory (ROM), a rigid magnetic disc, and an optical disk, such as a CD-ROM, CD-R/W or DVD.

[0034] In an implementation, the modules, components and other features described herein (for example control unit 110 in relation to Figure 1) can be implemented as discrete components or integrated in the functionality of hardware components such as ASICS, FPGAs, DSPs or similar devices as part of an individualization server.

[0035] A "hardware component" is a tangible (e.g., non-transitory) physical component (e.g., a set of one or more processors) capable of performing certain operations and may be configured or arranged in a certain physical manner. A hardware component may include dedicated circuitry or logic that is permanently configured to perform certain operations. A hardware component may be or include a special-purpose processor, such as a field programmable gate array (FPGA) or an ASIC. A hardware component may also include programmable logic or circuitry that is temporarily configured by software to perform certain operations.

[0036] Accordingly, the phrase "hardware component" should be understood to encompass a tangible entity that may be physically constructed, permanently configured (e.g., hardwired), or temporarily configured (e.g., programmed) to operate in a certain manner or to perform certain operations described herein.

[0037] In addition, the modules and components can be implemented as firmware or functional circuitry within hardware devices. Further, the modules and components can be implemented in any combination of hardware devices and software components, or only in software (e.g., code stored or otherwise embodied in a machine-readable medium or in a transmission medium).

[0038] Unless specifically stated otherwise, as apparent from the following discussion, it is appreciated that throughout the description, discussions utilizing terms such as " receiving", "determining", "comparing ", "enabling", "maintaining," "identifying," "replacing," or the like, refer to the actions and processes of a computer system, or similar electronic computing device, that manipulates and transforms data represented as physical (electronic) quantities within the computer system's registers and memories into other data similarly represented as physical quantities within the computer system memories or registers or other such information storage, transmission or display devices.

[0039] It is to be understood that the above description is intended to be illustrative, and not restrictive. Many other implementations will be apparent to those of skill in the art upon reading and understanding the above description. Although the present disclosure has been described with reference to specific example implementations, it will be recognized that the disclosure is not limited to the implementations described, but can be practiced with modification and alteration within the spirit and scope of the appended claims. Accordingly, the specification and drawings are to be regarded in an illustrative sense

rather than a restrictive sense. The scope of the disclosure should, therefore, be determined with reference to the appended claims, along with the full scope of equivalents to which such claims are entitled.

Claims

1. A method for protecting code, comprising carrying out an object-to-object file transformation, the object-to-object transformation comprising:

   identifying code to be protected within an input object file for encryption;
   extracting the identified code to be protected;
   replacing the identified code to be protected within the input object file with a fake code to generate a first output object file; and
   injecting the code to be protected into a second output object file.

2. A method according to claim 1, further comprising consolidating the first output object file and the second output object file in to a final object file.

3. A method according to claim 1 or claim 2, further comprising, prior to the object-to-object transformation, carrying out a source-to-source transformation, the source-to-source transformation comprising:

   marking the code to be protected within an input source file; and
   providing additional code to the input source file to provide instructions for a later decryption operation.

4. A method according to claim 3, wherein the source-to-source transformation further comprises isolating the code to be protected.

5. A method according to any one of the preceding claims, wherein the object-to-object transformation further comprises:

   identifying relocation directions within the input object file;
   excluding the relocation directions from replacement by the fake code.

6. A method according to any one of the preceding claims, wherein the fake code is selected to resemble real code.

7. A method according to any one of the preceding claims, wherein the object-to-object transformation further comprises encrypting the code to be protected.

8. A computer program product comprising computer executable instructions for carrying out the method of any one of the preceding claims.

9. A system for protecting code, the system comprising a processor arranged to carry out an object-to-object file transformation, the object-to-object transformation comprising:

   identifying code to be protected within an input object file for encryption;
   extracting the identified code to be protected;
   replacing the identified code to be protected within the input object file with a fake code to generate a first output object file; and
   injecting the code to be protected into a second output object file.

10. A system according to claim 9, wherein the processor is further arranged to consolidate the first output object file and the second output object file in to a final object file.

11. A system according to claim 9 or claim 10, wherein the processor is further arranged to, prior to the object-to-object transformation, carry out a source-to-source transformation, the source-to-source transformation comprising:

    marking the code to be protected within an input source file; and
    providing additional code to the input source file to provide instructions for a later decryption operation.

12. A system according to claim 11, wherein the source-to-source transformation further comprises isolating the code to be protected.

13. A system according to any one of claims 9 to 12, wherein the object-to-object transformation further comprises:

    identifying relocation directions within the input object file;
    excluding the relocation directions from replacement by the fake code.

14. A system according to any one of claims 9 to 13, wherein the fake code is selected to resemble real code.

15. A system according to any one of claims 9 to 14, wherein the object-to-object transformation further comprises encrypting the code to be protected.

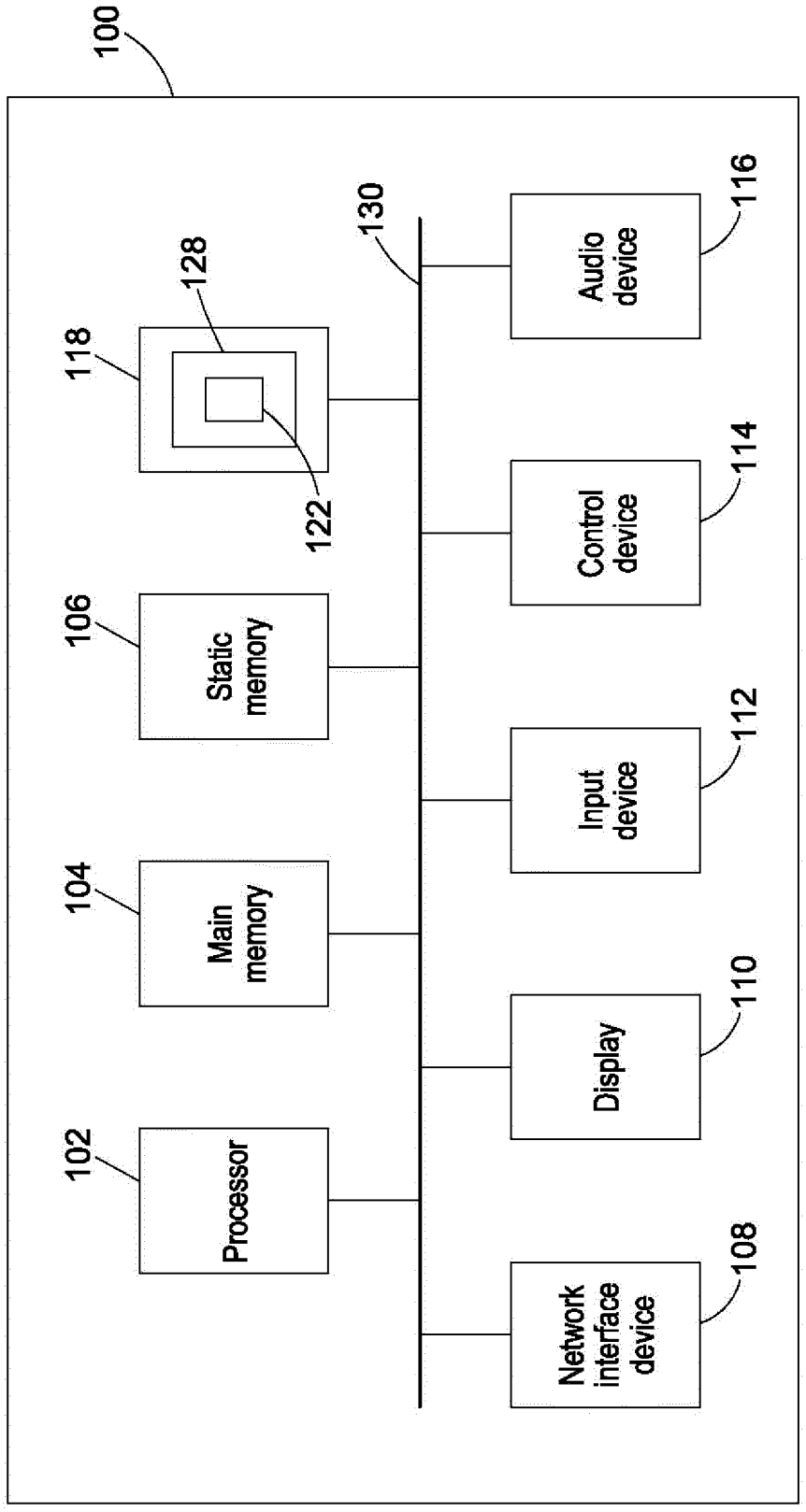
100
130
128
118
122
106
Static memory
104
Main memory
102
Processor
Audio device
116
Control device
114
Input device
112
Display
110
Network interface device
108

Fig. 1

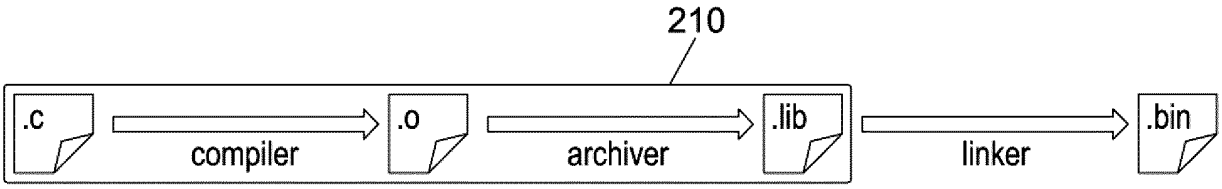

Fig. 2A

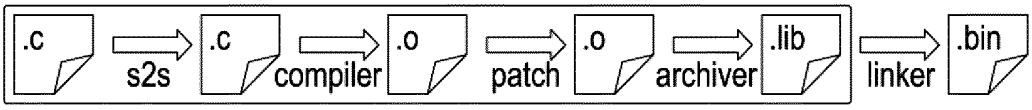

Fig. 2B

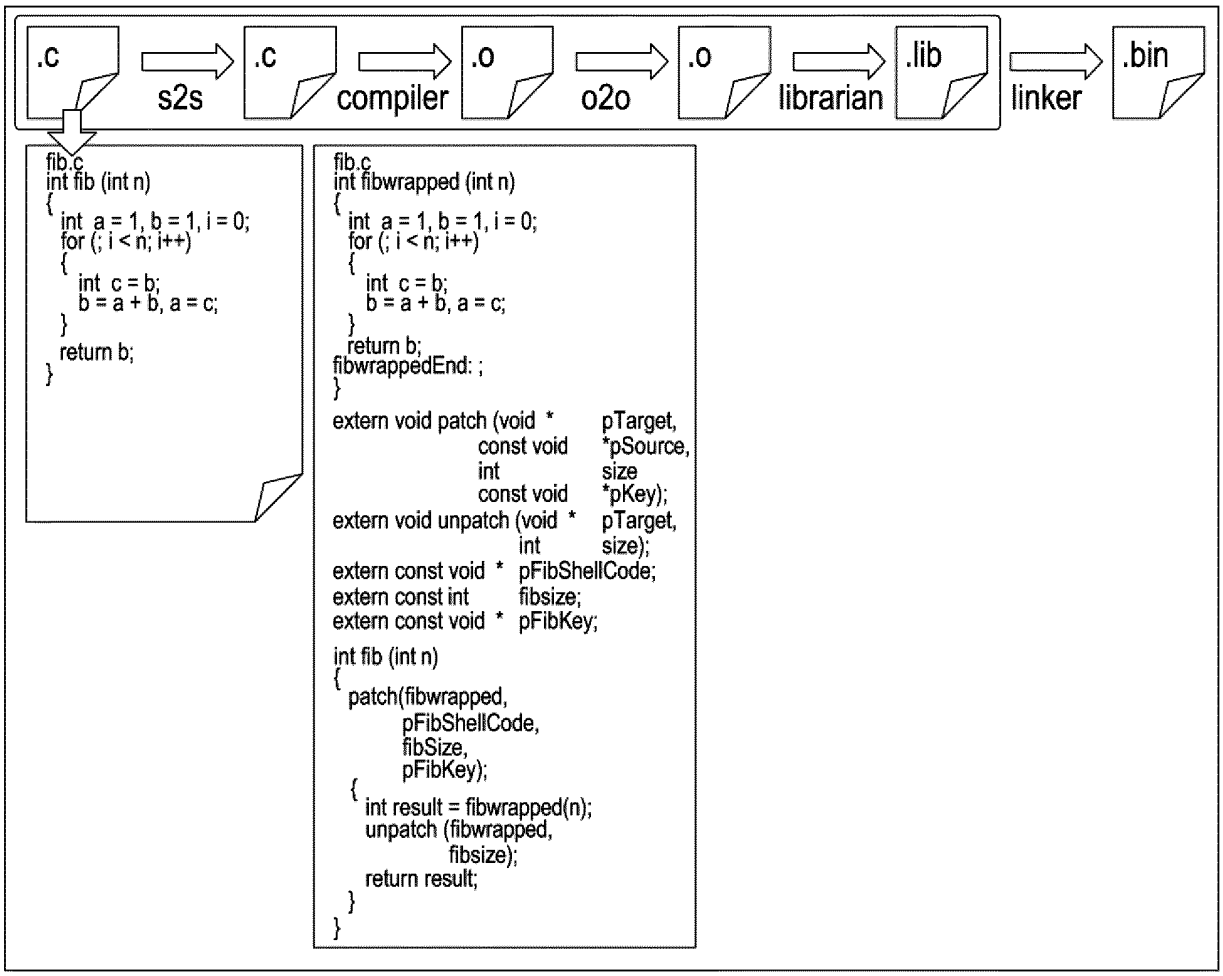

Fig. 3

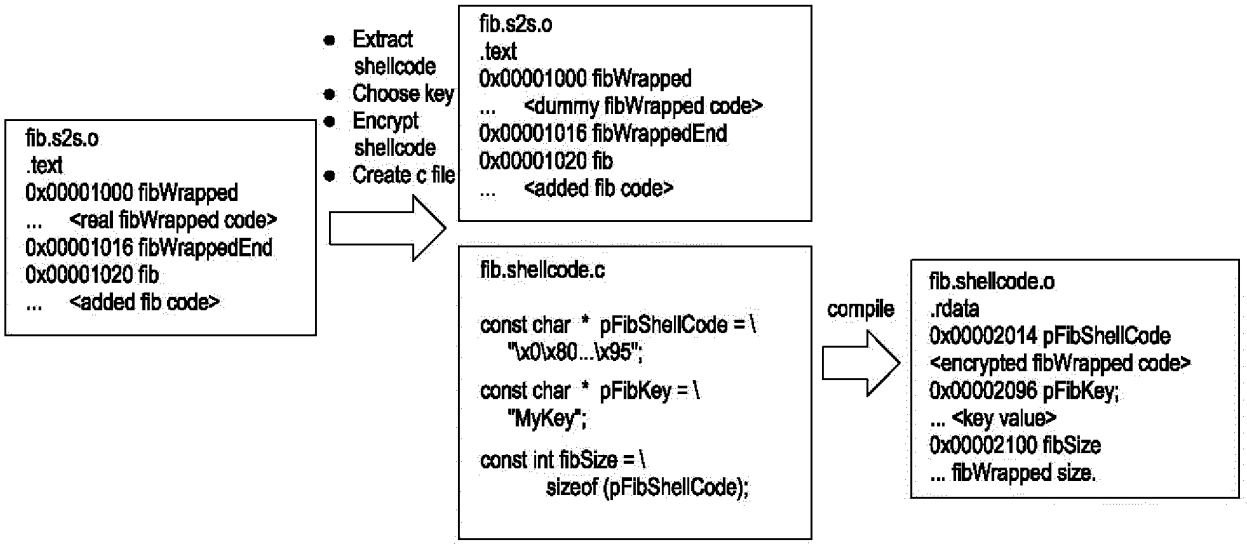
fib.s2s.o
.text
0x00001000 fibWrapped
... <real fibWrapped code>
0x00001016 fibWrappedEnd
0x00001020 fib
... <added fib code>
Extract shellcode
Choose key
Encrypt shellcode
Create c file
fib.s2s.o
.text
0x00001000 fibWrapped
... <dummy fibWrapped code>
0x00001016 fibWrappedEnd
0x00001020 fib
... <added fib code>
fib.shellcode.c
const char * pFibShellCode = \
"\x0\x80...\x95";
const char * pFibKey = \
"MyKey";
const int fibSize = \
sizeof (pFibShellCode);
compile
fib.shellcode.o
.rdata
0x00002014 pFibShellCode
<encrypted fibWrapped code>
0x00002096 pFibKey;
... <key value>
0x00002100 fibSize
... fibWrapped size.

Fig. 4

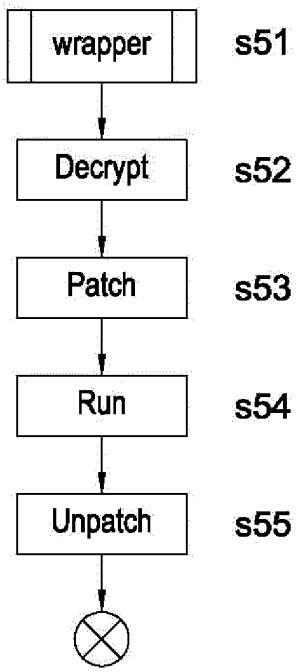
wrapper
s51
Decrypt
s52
Patch
s53
Run
s54
Unpatch
s55

Fig. 5

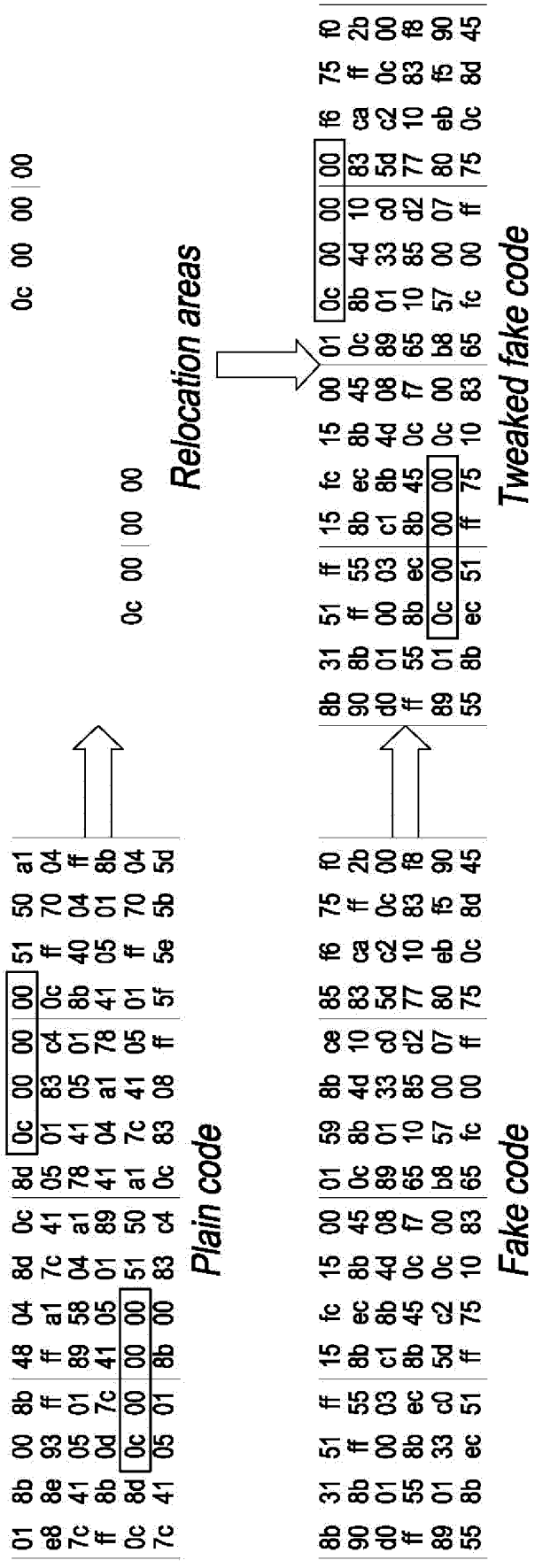

Fig. 6

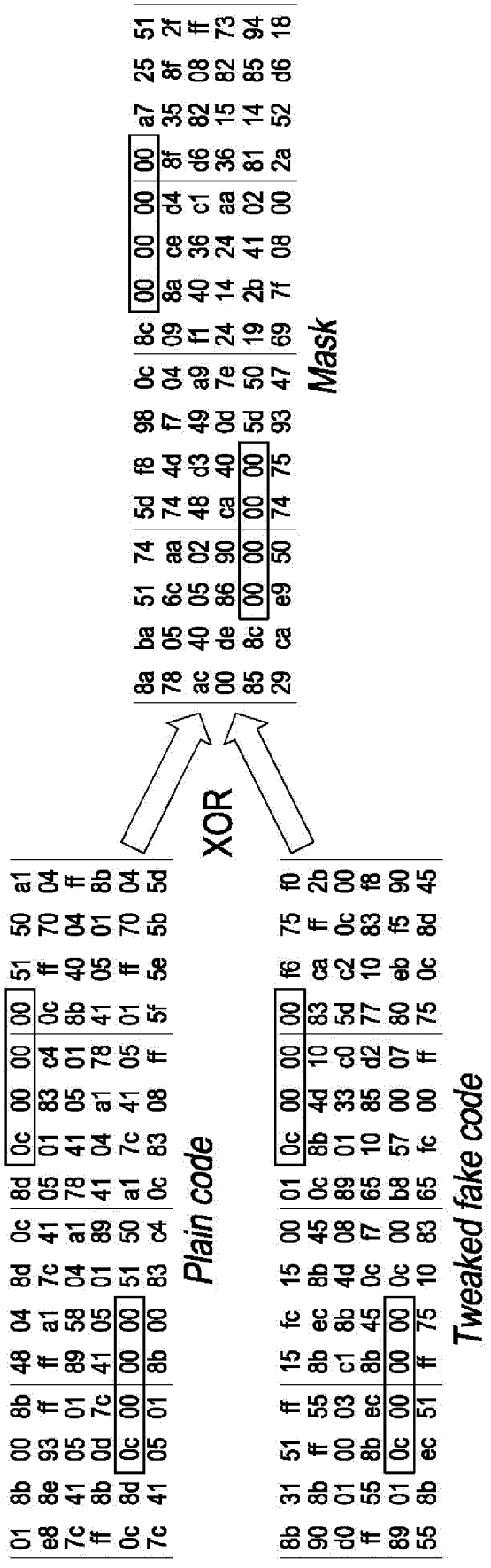

Fig. 7

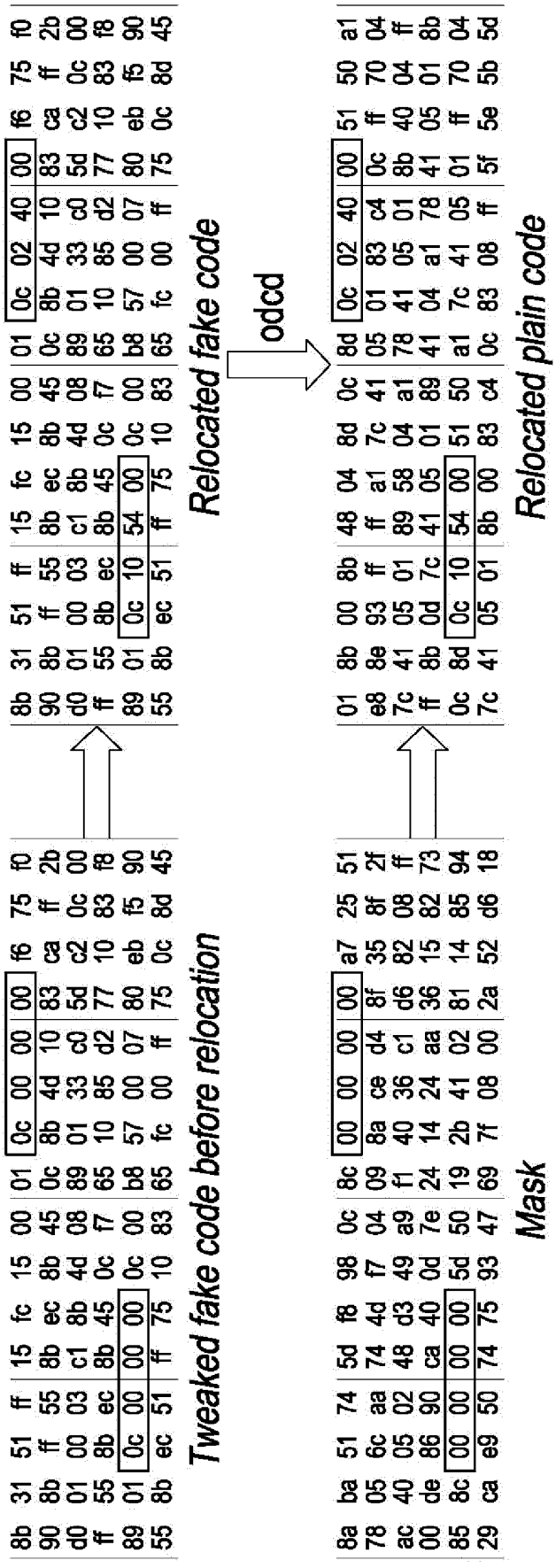
Tweaked fake code before relocation
Relocated fake code
odcd
Mask
Relocated plain code

Fig. 8

Europäisches Patentamt
European Patent Office
Office européen des brevets

EUROPEAN SEARCH REPORT

Application Number
EP 16 30 5797

DOCUMENTS CONSIDERED TO BE RELEVANT

Category	Citation of document with indication, where appropriate, of relevant passages	Relevant to claim	CLASSIFICATION OF THE APPLICATION (IPC)
X	US 2015/363580 A1 (BETOUIN PIERRE [FR] ET AL) 17 December 2015 (2015-12-17) * paragraphs [0026], [0045], [0053] *	1-15	INV. G06F21/14
Y	US 2016/092871 A1 (GORDON JAMES [US] ET AL) 31 March 2016 (2016-03-31) * abstract; figures 5,8 *	1-15	
Y	US 2012/260102 A1 (ZAKS GANNA [US] ET AL) 11 October 2012 (2012-10-11) * abstract; figures 5-7 *	1-15	
Y	WO 2007/063433 A2 (NXP BV [NL]; KING COLIN [GB]) 7 June 2007 (2007-06-07) * abstract; figures 4,6 *	1-15	

TECHNICAL FIELDS SEARCHED (IPC)

G06F

The present search report has been drawn up for all claims

Place of search	Date of completion of the search	Examiner
Munich	16 December 2016	Kerschbaumer, J

CATEGORY OF CITED DOCUMENTS

X : particularly relevant if taken alone
Y : particularly relevant if combined with another document of the same category
A : technological background
O : non-written disclosure
P : intermediate document

T : theory or principle underlying the invention
E : earlier patent document, but published on, or after the filing date
D : document cited in the application
L : document cited for other reasons

& : member of the same patent family, corresponding document

EPO FORM 1503 03.82 (P04C01)

ANNEX TO THE EUROPEAN SEARCH REPORT ON EUROPEAN PATENT APPLICATION NO. EP 16 30 5797

This annex lists the patent family members relating to the patent documents cited in the above-mentioned European search report. The members are as contained in the European Patent Office EDP file on
The European Patent Office is in no way liable for these particulars which are merely given for the purpose of information.

16-12-2016

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
US 2015363580 A1	17-12-2015	NONE	
US 2016092871 A1	31-03-2016	NONE	
US 2012260102 A1	11-10-2012	NONE	
WO 2007063433 A2	07-06-2007	CN 101288083 A	15-10-2008
		EP 1943607 A2	16-07-2008
		JP 2009512087 A	19-03-2009
		US 2009232304 A1	17-09-2009
		WO 2007063433 A2	07-06-2007

EPO FORM P0459

For more details about this annex : see Official Journal of the European Patent Office, No. 12/82